Linux 线程的实质

线程与进程的比较

对比维度	多进程	多线程	总结
数据共享、同步	数据共享复杂，需要用IPC；数据是分开的，同步简单	因为共享进程数据，数据共享简单，但也是因为这个原因导致同步复杂	各有优势
内存、CPU	占用内存多，切换复杂，CPU利用率低	占用内存少，切换简单，CPU利用率高	线程占优
创建销毁、切换	创建销毁、切换复杂，速度慢	创建销毁、切换简单，速度很快	线程占优
编程、调试	编程简单，调试简单	编程复杂，调试复杂	进程占优
可靠性	进程间不会互相影响	一个线程挂掉将导致整个进程挂掉	进程占优
分布式	适应于多核、多机分布式；如果一台机器不够，扩展到多台机器比较简单	适应于多核分布式	进程占优

概述:

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，进程是系统进行资源分配和调度的一个独立单位。

线程是进程的一个实体，是 CPU 调度和分派的基本单位，它是比进程更小的能独立调度的基本单位。线程自己基本上不拥有系统资源，只拥有一点在运行中必不可少的资源(如程序计数器，一组寄存器和栈)，但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。一个线程可以创建和撤销另一个线程；同一个进程中的多个线程之间可以并发执行。

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。在串行程序基础上引入线程和进程是为了提高程序的并发度，从而提高程序运行效率和响应时间。

区别:

进 程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个 进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线

程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

1) 简而言之，一个程序至少有一个进程，一个进程至少有一个线程。

2) 线程的划分尺度小于进程，使得多线程程序的并发性高。

3) 另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

4) 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

5) 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

优缺点:

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在 SMP 机器上运行，而进程可以跨机器迁移。

多进程，多线程

概述:

进程就是一个程序运行的时候被 CPU 抽象出来的，一个程序运行后被抽象为一个进程，但是线程是从一个进程里面分割出来的，由于 CPU 处理进程的时候是采用时间片轮转的方式，所以要把一个大个进程给分割成多个线程，例如：网际快车中文件分成 100 部分 10 个线程 文件就被分成了 10 份来同时下载 1-10 占一个线程 11-20 占一个线程，依次类推，线程越多，文件就被分的越多，同时下载 当然速度也就越快

进程是程序在计算机上的一次执行活动。当 你运行一个程序，你就启动了一个进程。显然，程序只是一组指令的有序集合，它本身没有任何运行的含义，只是一个静态实体。而进程则不同，它是程序在某个数 据集上的执行，是一个动态实体。它因创建而产生，因调度而运行，因等待资源或事件而被处于等待状态，因完成任务而被撤消，反映了一个程序在一定的数据集上 运行的全部动态过程。进程是操作系统分配资源的单位。在 Windows 下，进程又被细化为线程，也就是一个进程下有多个能独立运行的更小的单位。线程(Thread)是

进程的一个实体，是 CPU 调度和分派的基本单位。线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

线程和进程的关系是：线程是属于进程的，线程运行在进程空间内，同一进程所产生的线程共享同一内存空间，当进程退出时该进程所产生的线程都会被强制退出并清除。线程可与属于同一进程的其它线程共享进程所拥有的全部资源，但是其本身基本上不拥有系统资源，只拥有一点在运行中必不可少的信息(如程序计数器、一组寄存器和栈)。

在同一个时间里，同一个计算机系统中如果允许两个或两个以上的进程处于运行状态，这便是多任务。现代的操作系统几乎都是多任务操作系统，能够同时管理多个进程的运行。 多任务带来的好处是明显的，比如你可以边听 mp3 边上网，与此同时甚至可以将下载的文档打印出来，而这些任务之间丝毫不会相互干扰。那么这里就涉及到并行的问题，俗话说，一心不能二用，这对计算机也一样，原则上一个 CPU 只能分配给一个进程，以便运行这个进程。我们通常使用的计算机中只有一个 CPU，也就是说只有一颗心，要让它一心多用，同时运行多个进程，就必须使用并发技术。实现并发技术相当复杂，最容易理解的是“时间片轮转进程调度算法”，它的思想简单介绍如下：在操作系统的管理下，所有正在运行的进程轮流使用 CPU，每个进程允许占用 CPU 的时间非常短(比如 10 毫秒)，这样用户根本感觉不出来 CPU 是在轮流为多个进程服务，就好象所有的进程都在不间断地运行一样。但实际上在任何一个时间内有且仅有一个进程占有 CPU。

如果一台计算机有多个 CPU，情况就不同了，如果进程数小于 CPU 数，则不同的进程可以分配给不同的 CPU 来运行，这样，多个进程就是真正同时运行的，这便是并行。但如果进程数大于 CPU 数，则仍然需要使用并发技术。在 Linux 中，进行 CPU 分配是以 task 为单位的，一个进程可能由多个线程组成，这时情况更加复杂，但简单地说，有如下关系：

总 task 数<= CPU 数量：并行运行

总 task 数> CPU 数量：并发运行

并行运行的效率显然高于并发运行，所以在多 CPU 的计算机中，多任务的效率比较高。但是，如果在多 CPU 计算机中只运行一个进程(线程)，就不能发挥多 CPU 的优势。

多任务操作系统(如 Windows)的基本原理是:操作系统将 CPU 的时间片分配给多个线程，每个线程在操作系统指定的时间片内完成(注意，这里的多个线程是分属于不同进程的)。操作系统不断的从一个线程的执行切换到另一个线程的执行，如此往复，宏观上看来，就好像是多个线程在一起执行。由于这多个线程分属于不同的进程，因此在我们看来，就好像是多个进程在同时执行，这样就实现了多任务。

分类

根据进程与线程的设置，操作系统大致分为如下类型：

(1) 单进程、单线程，MS-DOS 大致是这种操作系统；

(2) 多进程、单线程，多数 UNIX(及类 UNIX 的 LINUX)是这种操作系统；

(3) 多进程、多线程，Win32(Windows NT/2000/XP 等)、Solaris 2。x 和 OS/2 都是这种操作系统；

(4) 单进程、多线程，VxWorks 是这种操作系统。

引入线程带来的主要好处

(1)在进程内创建、终止线程比创建、终止进程要快；

(2)同一进程内的线程间切换比进程间的切换要快，尤其是用户级线程间的切换。

(3)线程间通信效率比进程间通信效率高。

用户级线程 (UserLevel Threads — ULT)

内核级线程 (Kernel Supported threads — KST)

内核级线程 KST

对于一切的进程，无论是系统进程还是用户进程，进程的创建和撤销，以及 I/O 操作都是利用系统调用进入到内核，由内核处理完成，所以说在 KST 下， 所有进程都是在操作系统内核的支持下运行的，是与内核紧密相关的。内核空间实现还为每个内核支持线程设置了一个线程控制快，内核是根据该控制快而感知某个线程是否存在，并加以控制。

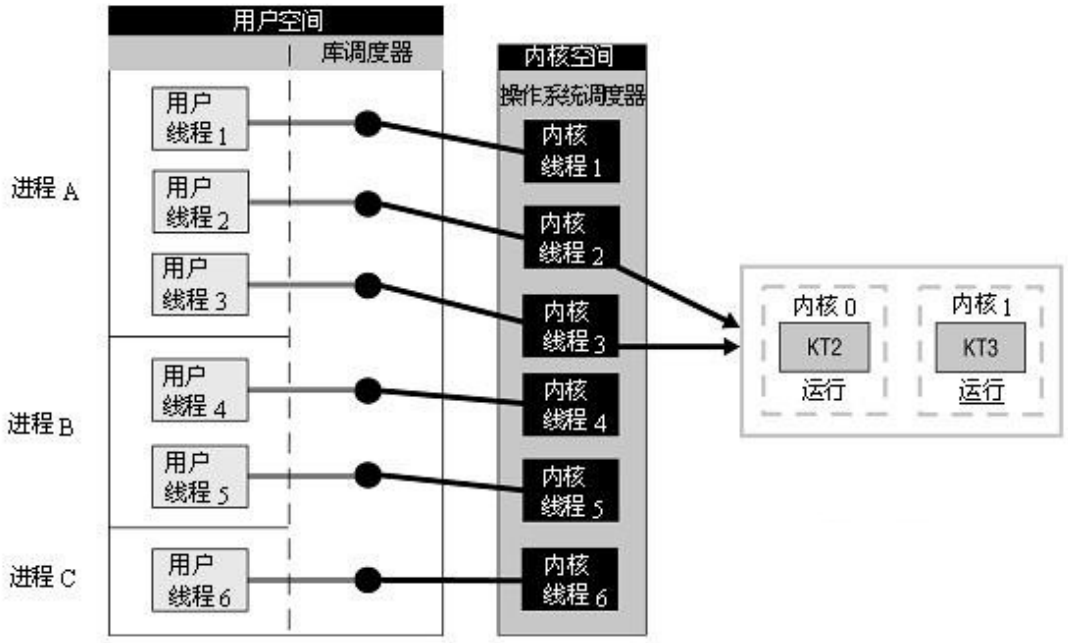
优点：

1．在多处理器上，内核可以调用同一进程中的多个线程同时工作；

2． 如果一个进程中的一个线程阻塞了，其他线程仍然可以得到运行；

缺点：对于用户线程的切换代价太大，在同一个线程中，从一个线程切换到另一个线程时，需要从用户态，进入到内核态并且由内核切换。因为线程调度和管理在内核实现。

内核级线程驻留在内核空间，它们是内核对象。有了内核线程，每个用户线程被映射或绑定到一个内核线程。用户线程在其生命期内都会绑定到该内核线程。一旦用 户线程终止，两个线程都将离开系统。这被称作"一对一"线程映射，如下图所示。操作系统调度器管理、调度并分派这些线程。运行时库为每个用户级 线程请求一个内核级线程。操作系统的内存管理和调度子系统必须要考虑到数量巨大的用户级线程。您必须了解每个进程允许的线程的最大数目是多少。操作系统为 每个线程创建上下文。进程的每个线程在资源可用时都可以被指派到处理器内核。



用户级线程 ULT

用户进程 ULT 仅存在于用户空间中。对于这种线程的创建、撤销、线程之间的同步和通信等功能，都无需系统调用来实现。对于同一进程的线程之间切换仍然是不需要内核支持的。所以，内核也会是完全不会知道用户级线程的存在。

但是有一点必须注意：设置了用户级线程的系统，其调度茬苒是以进程为单位进行的哦。

优点：

1。 线程切换不需要转换到内核空间，节省了宝贵的内核空间；

2。 调度算法可以是进程专用，由用户程序进行指定；

3。 用户级线程实现和操作系统无关；

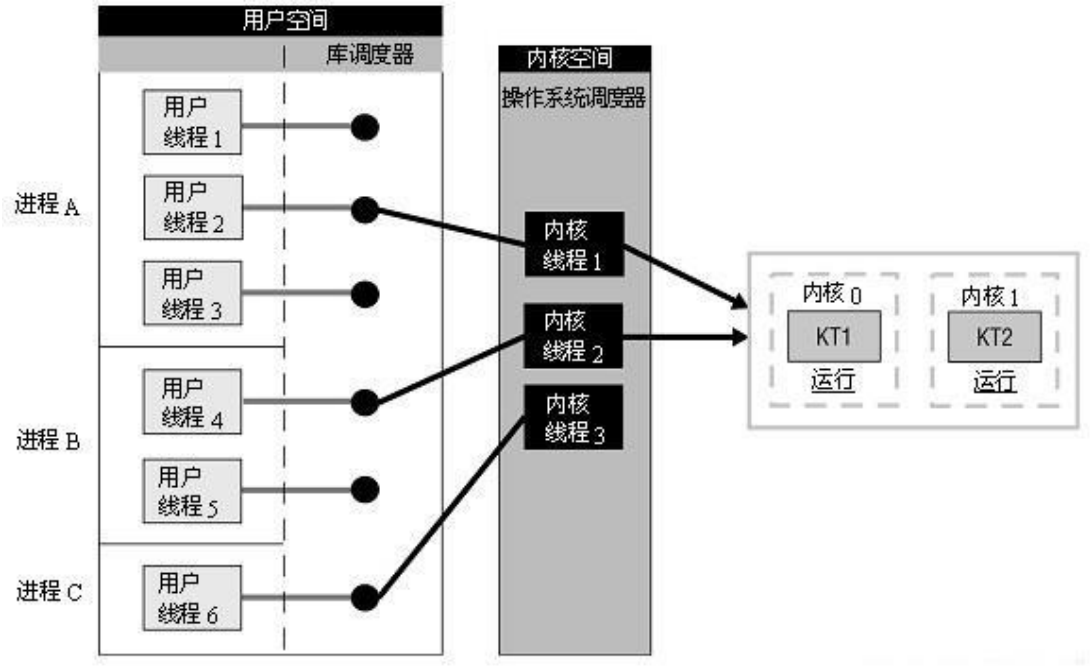
缺点:

1。 系统调用阻塞，同一进程中一个线程阻塞和整个进程都阻塞了。

2。 一个进程只能在一个 cpu 上获得执行。

用户级线程驻留在用户空间或模式。运行时库管理这些线程，它也位于用户空间。它们对于操作系统是不可见的，因此无法被调度到处理器内核。每个线程并不具有 自身的线程上下文。因此，就线程的同时执行而言，任意给定时刻每个进程只能够有一个线程在运行，而且只有一个处理器内核会被分配给该进程。对于一个进程， 可能有成千上万个用户级线程，但是它们对系统资源没有影响。运行时库调度并分派这些线程。如同在图中看到的那样，库调度器从进程的多个线程中选择一个线程，然后该线程和该进程允许的一个内核线程关联起来。内核线程将被操作系统调度器指派到处理器内核。用户级线程是一种"多对一"的线程

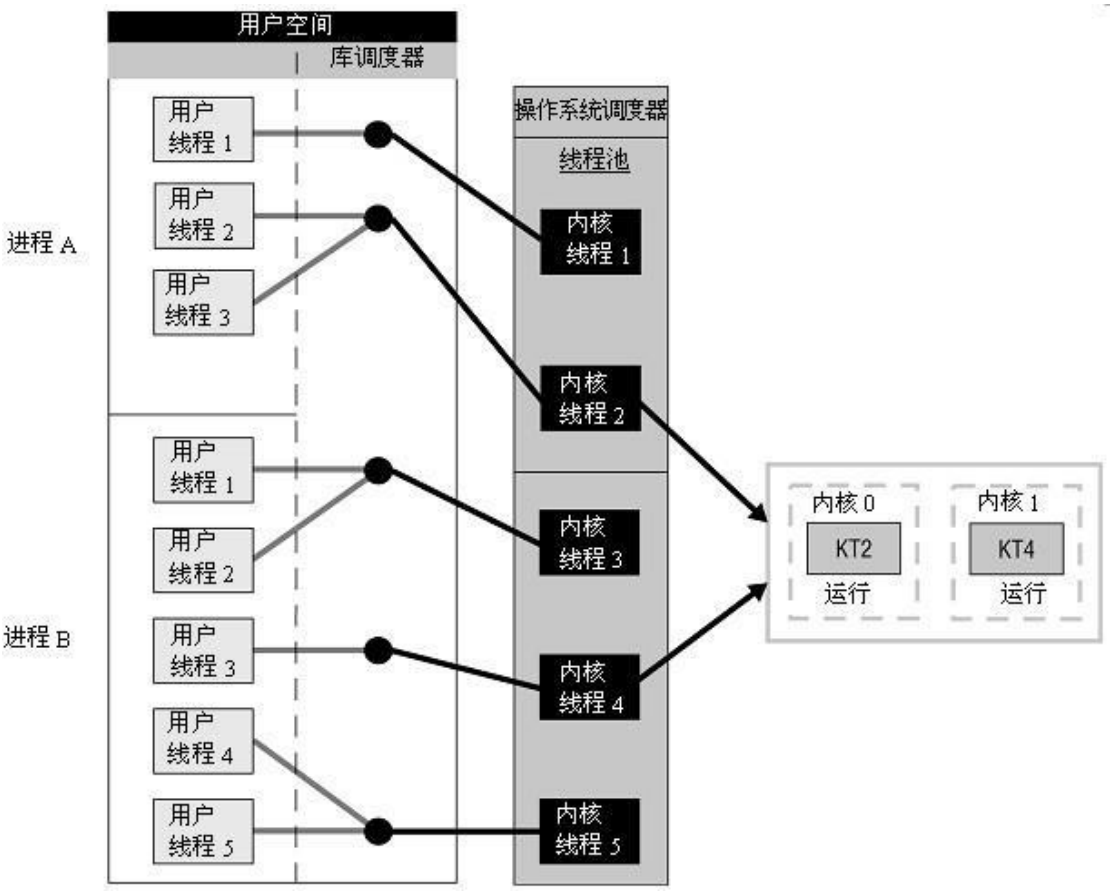
映射。



混合方式

在很多的操作系统中 ULT 和 KLT 进行组合，整合了 ULT 和 KLT 的优点。混合线程实现是用户线程和内核线程的交叉，使得库和操作系统都可以管理线程。用户线程由运行时库调度器管理，内核线程由操作系统调度器管理。在这种实现 中，进程有着自己的内核线程池。可运行的用户线程由运行时库分派并标记为准备好执行的可用线程。操作系统选择用户线程并将它映射到线程池中的可用内核线 程。多个用户线程可以分配给相同的内核线程。在图中，进程 A 在它的线程池中有两个内核线程，而进程 B 有 3 个内核线程。进程 A 的用户线程 2 和 3 被映射到内核线程(2)。进程 B 有 5 个线程，用户线程 1 和 2 映射到同一个内核线程(3)，用户线程 4 和 5 映射到内核同一个内核线程(5)。当创建新的用户 线程时，只需要简单地将它映射到线程池中现有的一个内核线程即可。这种实现使用了"多对多"线程映射。该方法中尽量使用多对一映射。很多用户线程将会映射 到一个内核线程，就像您在前面的示例中所看到的。因此，对内核线程的请求将会少于用户线程的数目。

内核线程池不会被销毁和重建，这些线程总是位于系统中。它们会在必要时分配给不同的用户级线程，而不是当创建新的用户级线程时就创建一个新的内核线程，而 内核支持线程被创建时，就会创建一个新的内核线程。只对池中的每个线程创建上下文。有了内核线程和混合线程，操作系统分配一组处理器内核，进程的线程可以 在这些处理器内核之上运行。线程只能在为它们所属线程指派的处理器内核上运行。



线程上下文

操作系统管理很多进程的执行。有些进程是来自各种程序、系统和应用程序的单独进程，而某些进程来自被分解为很多进程的应用或程序。当一个进程从内核中移出，另一个进程成为活动的，这些进程之间便发生了上下文切换。操作系统必须记录重启进程和启动新进程使之活动所需要的所有信息。这些信息被称作上下 文，它描述了进程的现有状态。当进程成为活动的，它可以继续从被抢占的位置开始执行。进程的上下文信息包括：进程 id、指向可执行文件的指针、栈、静态和动态分配的变量的内存、处理器寄存器。进程上下文的多数信息都与地址空间的描述有关。进程的上下文使用很多系统资源，而且会花费一些时间来从一个进程的上下文切换到另一个进程的上下文。 线程也有上下文，当线程被抢占时，就会发生线程之间的上下文切换。如果线程属于相同的进程，它 们共享相同的地址空间，因为线程包含在它们所属于的进程的地址空间内。这样，进程需要恢复的多数信息对于线程而言是不需要的。尽管进程和它的线程共享了很 多内容，但最为重要的是其地址空间和资源，有些信息对于线程而言是本地且唯一的，而线程的其他方面包含在进程的各个段的内部。

上下文内容	进 程	线 程
指向可执行文件的指针	x	
栈	x	x
内存(数据段和堆)	x	
状态	x	x
优先级	x	x
程序 I/O 的状态	x	
授予权限	x	
调度信息	x	
审计信息	x	
有关资源的信息 ● 文件描述符 ● 读/写指针	x	
有关事件和信号的信息	x	
寄存器组 ● 栈指针 ● 指令计数器 ● 诸如此类	x	x

线程本地（且唯一）的信息包括线程 id、处理器寄存器(当线程执行时寄存器的状态，包括程序计数器和栈指针)、线程状态及优先级、线程特定数 据(thread-specific data，TSD)。线程 id 是在创建线程时指定的。线程能够访问它所属进程的数据段，因此线程可以读写它所属进程的全局声明数据。进程中一个线程做出的 任何改动都可以被进程中的所有线程以及主线程获得。在多数情况下，这要求某种类型的同步以防止无意的更新。线程的局部声明变量不应当被任何对等线程访问。 它们被放置到线程栈中，而且当线程完成时，它们便会被移走。